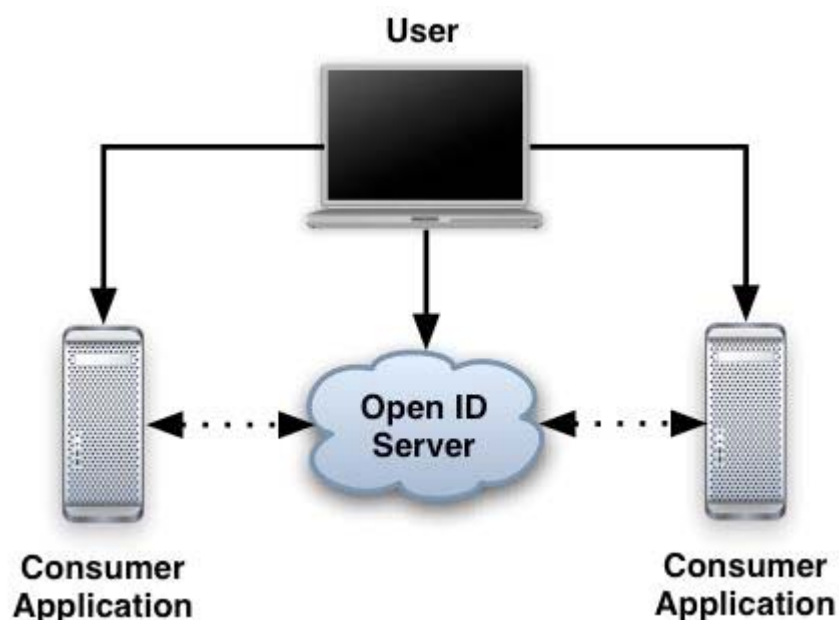


A Primer for OpenID with PHP

by Jack Herrington
September 21 2007

OpenID is another one of those "cool technologies you've never heard of" type of deals. OpenID is an open source initiative that provides a way for Web users to register their identity in one place and then use that identity anywhere on the Web that supports OpenID. This means that, as a user, you don't have to keep creating (and remembering) new user names and passwords at every site you visit. And, as a service provider, you can identify and authenticate users that show up at your site without having to do all of the user management involved in giving them access. The list of services that support OpenID is growing, but not as fast as it should be given how cool and useful OpenID is. Hopefully, this article will help with that.

The idea behind OpenID is fairly simple: a Web user can get an ID from an OpenID provider and use that ID to access any Web application or service that supports OpenID.



The user first creates an account with an OpenID provider. This could be a source like Verisign, or AOL, where all the AOL and AOL Instant Messenger (AIM) accounts are already OpenID-enabled. When the user has an OpenID they can log in to any OpenID-enabled "consumer" application on the Web. That is, a Web application that will accept OpenIDs as logins and use the corresponding OpenID provider to perform the required authentication. There is even a directory of Web applications that already take OpenIDs.

Becoming an OpenID consumer is really easy, as you will see from the code in this article. You can either use a canned library to do the work--these are available at no cost in all of the popular languages. Or you can, as I do in this article, write your own code. Although writing your own code leads to good understanding of the OpenID protocol, it's far more likely that in practice you will use one of the existing libraries to make it easy on yourself.

The advantages for both users and OpenID consumer applications are pretty compelling. Users can have a single OpenID and log in to lots of services. Which is really handy if you, like me, use a lot of Web applications and are constantly forgetting the user name and password for that particular site. The consumer application gets user management for free, because users simply use their OpenID. And consumer applications also get to jump into a huge pool of potential users simply by registering themselves as an OpenID consumer.

Before getting an OpenID, though, I should point out that it's not required that you run an OpenID provider to OpenID-enable your application. In fact, no special software or service is required in your Web application. The only requirement is that your Web server programming language be able to make HTTP requests of the OpenID provider, which is something all of them can do.

Getting an OpenID

Using OpenID starts by getting an OpenID from an OpenID provider. Fortunately, they are plentiful and free. I got the OpenID I use for this article from pip.verisignlabs.com. But you can get it from signon.com or wherever you like. An OpenID is actually a URL. The OpenID I used for this article is:

```
http://jherr.pip.verisignlabs.com
```

And that's not a fake URL; it's a real URL that returns an HTML page just like any other page. Embedded in the HTML for the page is a link to the provider that programs can use to verify a user's identity.

If I'm running UNIX I can just use `cURL` to get the OpenID URL and see what the HTML looks like. If I have only a browser I could go to the URL and view the source. Either way I get something like this:

```
% curl "http://jherr.pip.verisignlabs.com"
<html>
<head>
<link rel="openid.server" href="http://pip.verisignlabs.com/server" />
<meta http-equiv="X-XRDS-Location"
content="http://pip.verisignlabs.com/user/jherr/yadisxrds" />
<title>Identity Endpoint For jherr</title>
</head> <body>
<p>This is an identity endpoint for jherr</p>
<p>For more information, please visit
<a href="http://pip.verisignlabs.com">http://pip.verisignlabs.com</a>
</p></body></html>
%
```

The important bit is that `<link>` tag with the `rel` attribute of **openid.server**. That tells OpenID consumer sites (for example, blogs) that the provider that can authenticate my OpenID is located at that `href` value.

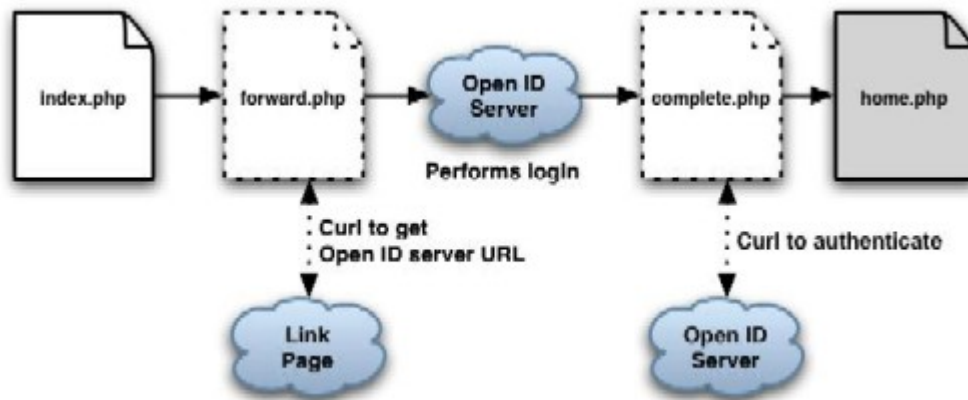
There are two key points so far:

1. Any page can be used as an OpenID URL as long as it provides the `<link>` tag with the `rel` attribute of **openid.server**. So you could use the landing page for your blog as your OpenID.
2. OpenID is a framework for managing online identities and not a single service provided by a specific company. So, lots of companies and organizations can and do provide free OpenIDs for which they handle all of the authentication.

Those are the essentials of OpenID from the user standpoint. To dig deeper we need see how to build an OpenID consumer application that users with OpenIDs can log in to, and how to build an OpenID provider application that vends and tracks OpenIDs.

Building a Consumer

With the basics of what an OpenID is out of the way it's time to build a simple OpenID consumer. The following diagram shows an OpenID page flow of an OpenID-enabled PHP site.



The line from left to right along the top is the flow of pages that the user would experience. The little blobs along the bottom indicate when your service accesses the OpenID services behind the scenes, where the user doesn't see it.

Starting on the far left the user goes to the login page for your site, Index.php. Here they are given a text field where they type their OpenID. The user clicks Submit, and the form goes to Forward.php. Forward.php and Complete.php are shown dotted because the user wouldn't normally see them.

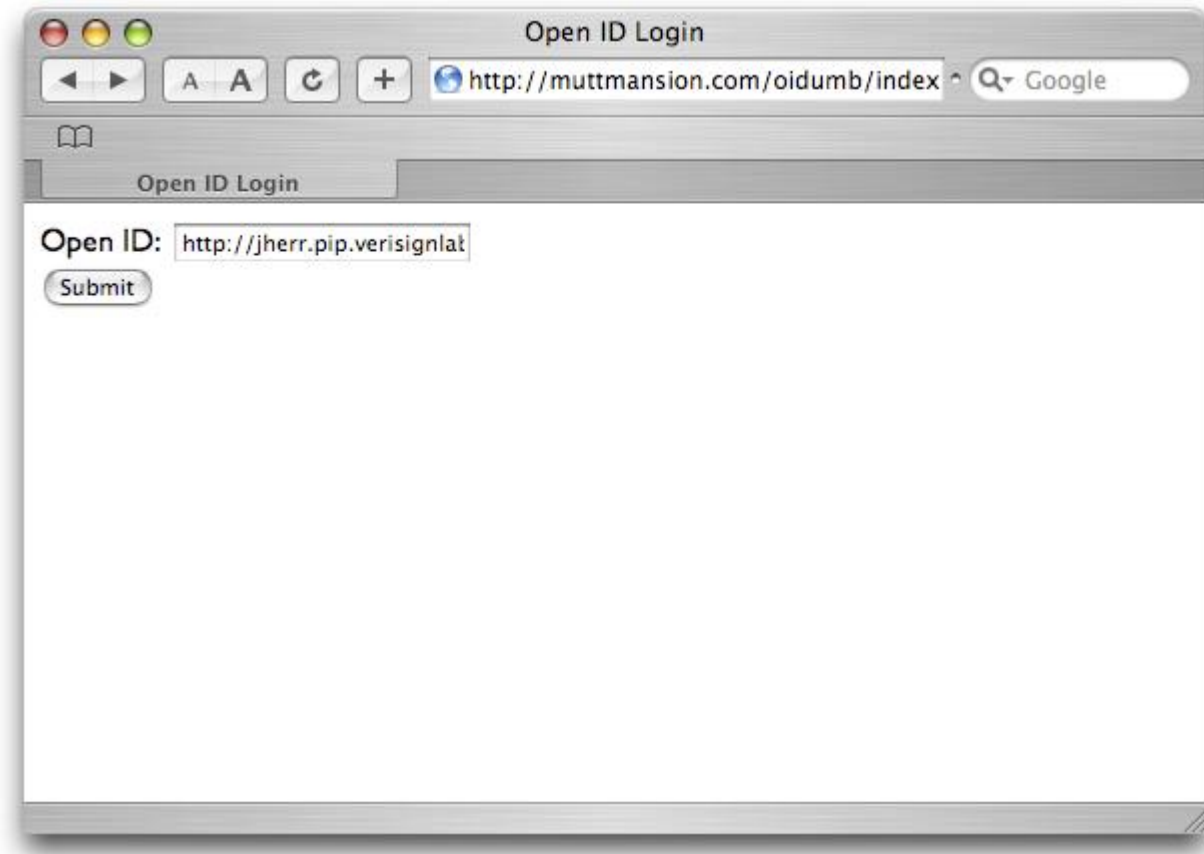
Forward.php takes the OpenID URL and uses cURL to retrieve the page. It then scans the page for the **openid.server** URL. With that URL in hand, it forwards the user to that OpenID provider URL, with some additional URL parameters.

The user is then forwarded to the OpenID provider, which does whatever it does: password, fob, blood sample, retinal scan, or whatever it takes to log you in. The OpenID provider then sends the OpenID back to the consumer service--in this example, the Complete.php URL--with some additional URL parameters.

Complete.php then goes back to the OpenID provider one more time to authenticate that it really was given the correct identity by the OpenID provider itself and not by some spoofing site. If that authentication works, the user is "logged in" and should be able to do whatever you allow them to do within the application or service.

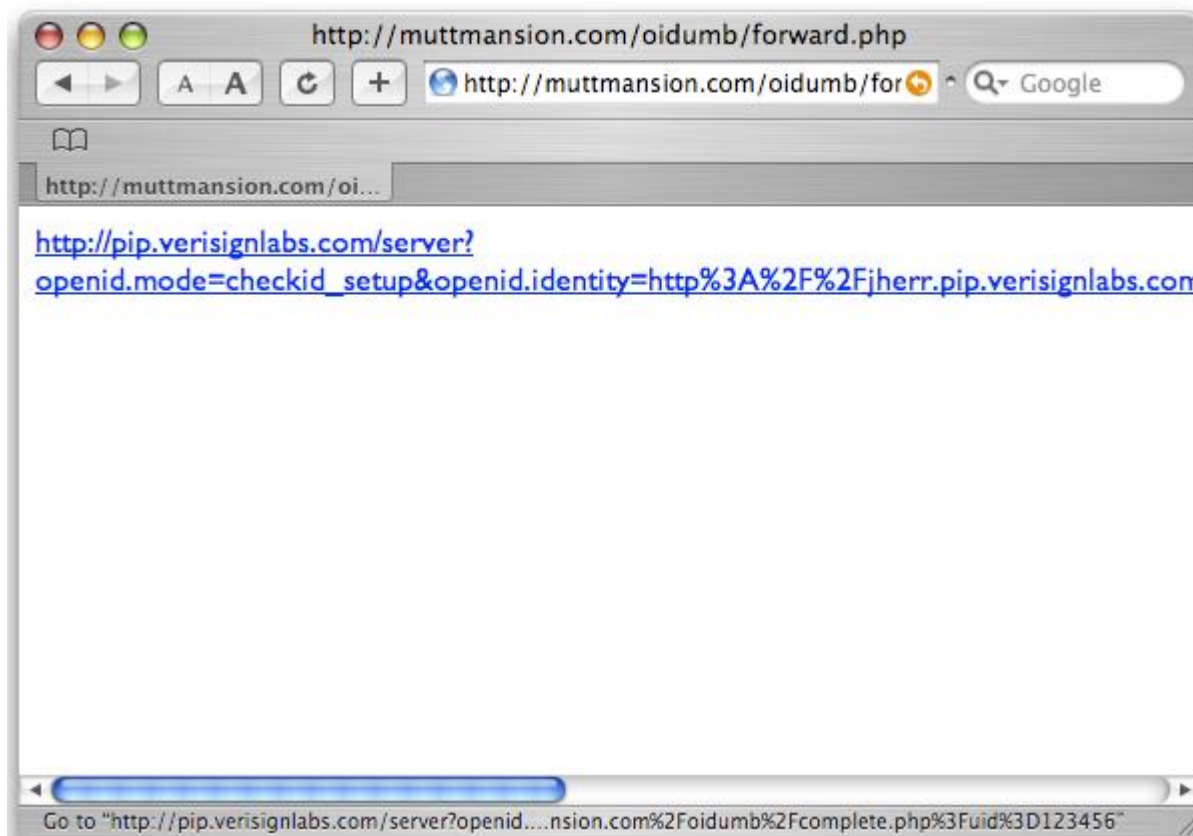
Complete.php can also verify that the response came from the OpenID provider itself without making another call to the OpenID provider. A consumer application can obtain a secret key from the OpenID provider before forwarding the user to the OpenID provider, and use that key to check the signature of the response from the OpenID provider. To keep the code simple in this sample application, this mode is not discussed in detail here.

I know this sounds kind of complex and convoluted. It's really not. Let's see what it looks like in the Web browser, beginning with the OpenID login page.



This couldn't be easier. It's a simple form with just one text entry field where I type my ID, and click Submit. It doesn't even have to be PHP!

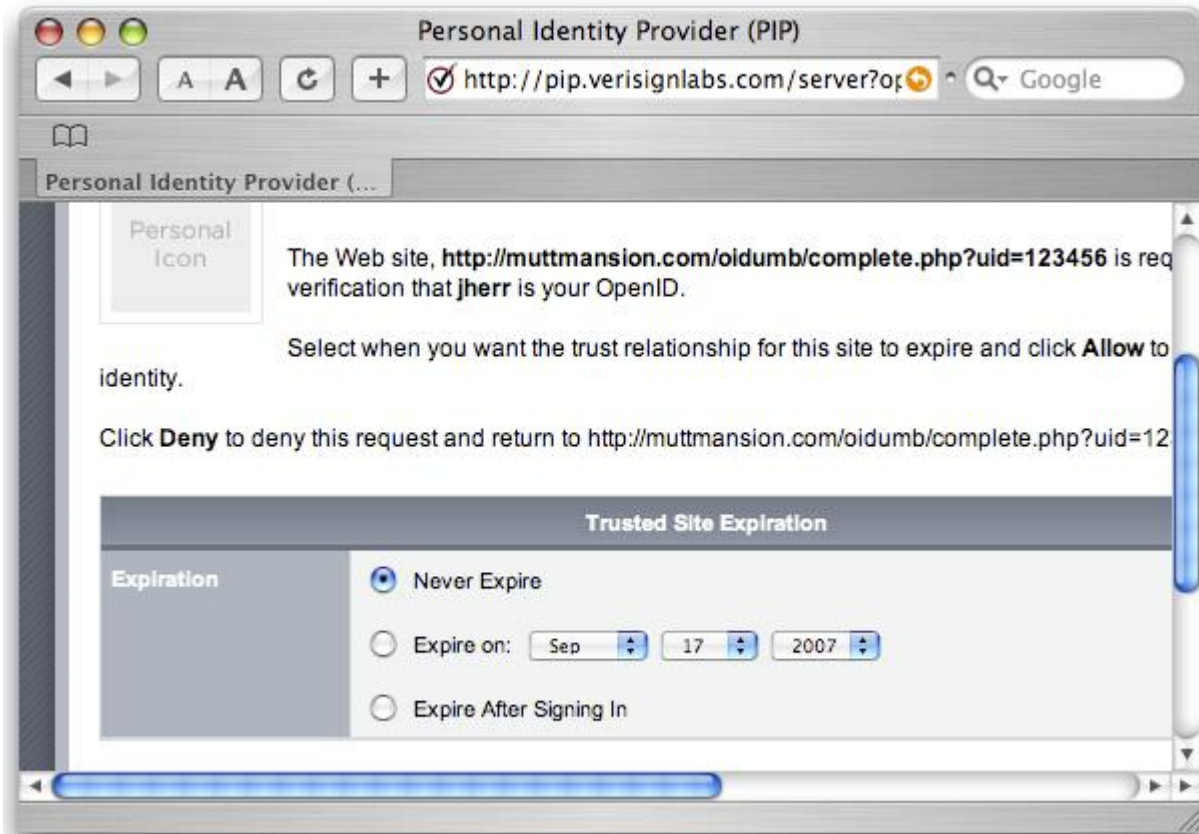
After clicking Submit, I see Forward.php.



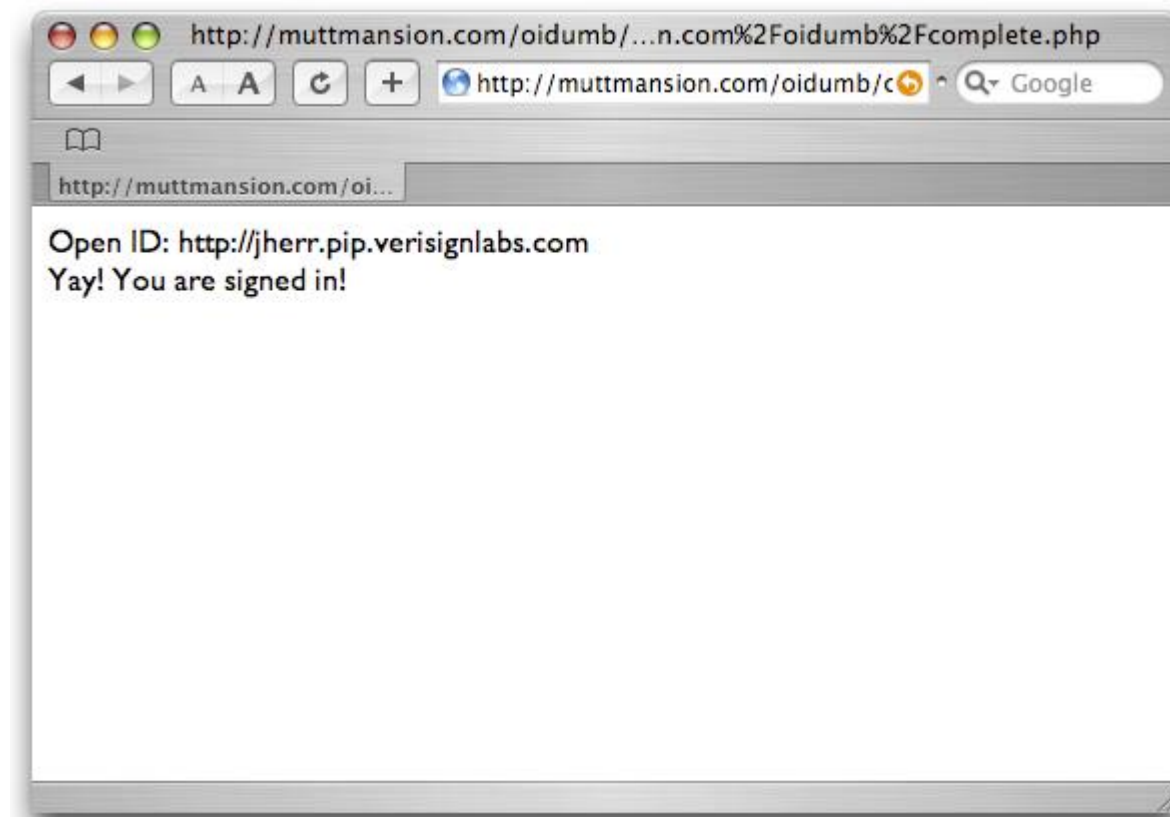
Normally the user won't see this page. The PHP would just forward them directly to the OpenID provider. But I used a link so that you can see what the URL that is sent to the OpenID provider looks like.

The arguments on the URL for the OpenID provider are pretty simple. The **openid.identity** key tells the OpenID provider what the OpenID being requested is. The **openid.mode** of **checkid_setup** tells the OpenID provider it should check the identity by trying to log them in (or whatever it does). And **openid.return_to** tells the OpenID provider where to send the user back to when they have successfully logged in.

Assuming that works, the next thing the user will see is a page from the OpenID provider.



Here I can specify how long the session should last, and so on. That's not really important to this example. When I click OK on this page, I'm sent back to Complete.php on the consumer site I am trying to access.



This little page, which says that I have successfully logged in, is actually deceptively simple. It's actually going back to the OpenID provider one last time using the OpenID “authenticate” mode just to make sure we aren't getting spoofed.

So, now that we have had a look at the high-level page flow, and looked at how all this appears to the user in the browser, it's time to look at the PHP code on our consumer site, which really does the work.

The Consumer Code

The consumer code starts with the OpenID login page, shown in Listing 1.

Listing 1. Index.php

```
<html><head><title>OpenID Login</title></head><body>
<form action="forward.php" method="post">
OpenID: <input type="text" name="url" /><br/>
<input type="submit" />
</form>
</body></html>
```

There is really nothing to it. It's just a standard HTML form with a text field and a Submit button that sends the data to the Forward.php page. Forward.php is shown in Listing 2.

Listing 2. Forward.php

```
<?php
ob_start();
$ch = curl_init();
curl_setopt($ch, CURLOPT_URL, $_REQUEST['url'] );
curl_setopt($ch, CURLOPT_HEADER, 0);
curl_exec($ch);
curl_close($ch);
$idhtml = ob_get_clean();

preg_match( "<link rel=\"openid.server\" href=\"(.*)\" />", $idhtml, $found );
$url = $found[1];

$return_to = "http://myhost.com/oidumb/complete.php";

$url .= "?openid.mode=checkid_setup";
$url .= "&openid.identity=".urlencode( $_REQUEST['url'] );
$url .= "&openid.return_to=".urlencode( $return_to );
?>
<html>
<body>
<a href="<?php echo($url);?>"><?php echo($url);?></a>
</body>
</html>
```

This page is a bit more complicated. It starts by using the cURL library, which is built into PHP, to retrieve the contents of the OpenID URL specified by the user. It then uses a really crude regular expression to get the **openid.server** value from the HTML.

With the OpenID provider URL in hand, I add the extra parameters required to set up the OpenID **checkid_setup** request and put together a link I can click. If I just wanted to forward the user to this URL I could use the **header** function and specify the **Location** as the **\$url** value that I built.

The last page in the example is Complete.php, which is the page that the OpenID provider sends the user back to, with some additional URL arguments, when they have successfully logged in to the OpenID provider site. The code for Complete.php is in Listing 3.

Listing 3. Complete.php

```
<?php
ob_start();
$ch = curl_init();
curl_setopt($ch, CURLOPT_URL, $_GET['openid_identity'] );
curl_setopt($ch, CURLOPT_HEADER, 0);
curl_exec($ch);
curl_close($ch);
```

```
$idhtml = ob_get_clean();

preg_match( "<link rel=\"openid.server\" href=\"(.*)\" />", $idhtml, $found );
$url = $found[1];

$url .= "?openid.mode=check_authentication";
$url .= "&openid.assoc_handle=".urlencode( $_GET['openid_assoc_handle'] );
$url .= "&openid.sig=".urlencode( $_GET['openid_sig'] );
$url .= "&openid.signed=".urlencode( $_GET['openid_signed'] );
$url .= "&openid.response_nonce=".urlencode( $_GET['openid_response_nonce'] );
$url .= "&openid.identity=".urlencode( $_GET['openid_identity'] );
$url .= "&openid.return_to=".urlencode( $_GET['openid_return_to'] );
$url .= "&openid.invalidate_handle=123456";

ob_start();
$ch = curl_init();
curl_setopt($ch, CURLOPT_URL, $url );

curl_exec($ch);
curl_close($ch);
$ca_resp = ob_get_clean();

$authentic = false;
if ( preg_match( '/is_valid:true/', $ca_resp ) )
    $authentic = true;
?>
<html><body>
<?php if( $authentic ) { ?>
OpenID: <?php echo( $_GET['openid_identity'] ) ?><br/>
Yay! You are signed in!
<?php } else { ?>
Your OpenID could not be verified.
<?php } ?>
</body>
</html>
```

The top of the script looks exactly like Forward.php because it is. We first have to get the URL for the OpenID provider from the user's OpenID URL. With that in hand, the script constructs a new **openid.mode=authenticate** URL with a bunch of parameters required by the OpenID specification.

The script then uses cURL to get the authentication result from the provider. This result is placed into **\$ca_resp**. If that textual response returns **is_valid:true**, then the script knows that the provider was the one that made the request and it can consider this OpenID URL authentic.

From there this little test site could set a session variable with the user's OpenID URL and use that as their identity throughout their session with the Web service.

That's about it for this OpenID "hello world" type of example. The value of an example like this is not that you would use this code directly. The regular expressions are too tight, there is no error checking, and on and on. The really important thing to get from this example is the page flow for how OpenID works at a basic level, so you can see what is handled by the OpenID consumer site and what is handled by the OpenID provider site. And reading about this in the OpenID spec left me bleary-eyed.

In the real world you would use an OpenID library suited to the development language of your choice to do the consumer or provider work for you. And that is what I will concentrate on the second half of this article.

But, before we get into that let's just recap the basics.

1. OpenID is not a service but a protocol that allows "consumers" of OpenIDs to validate those IDs against a corresponding OpenID "provider"
2. There are lots of OpenID provider sites, where you can get your own personal OpenID, or you could run your own OpenID provider application just for you.
3. OpenID is really good when you care only about identifying users and not actually managing user accounts.
4. The OpenID page flow for the user bounces back and forth between the consumer site, which requests OpenID validation, and the provider site, which actually does the validation and returns the user to where they left off in the consumer service.
5. The consumer portion of the OpenID work flow is easy to write in any programming language that supports Web requests.

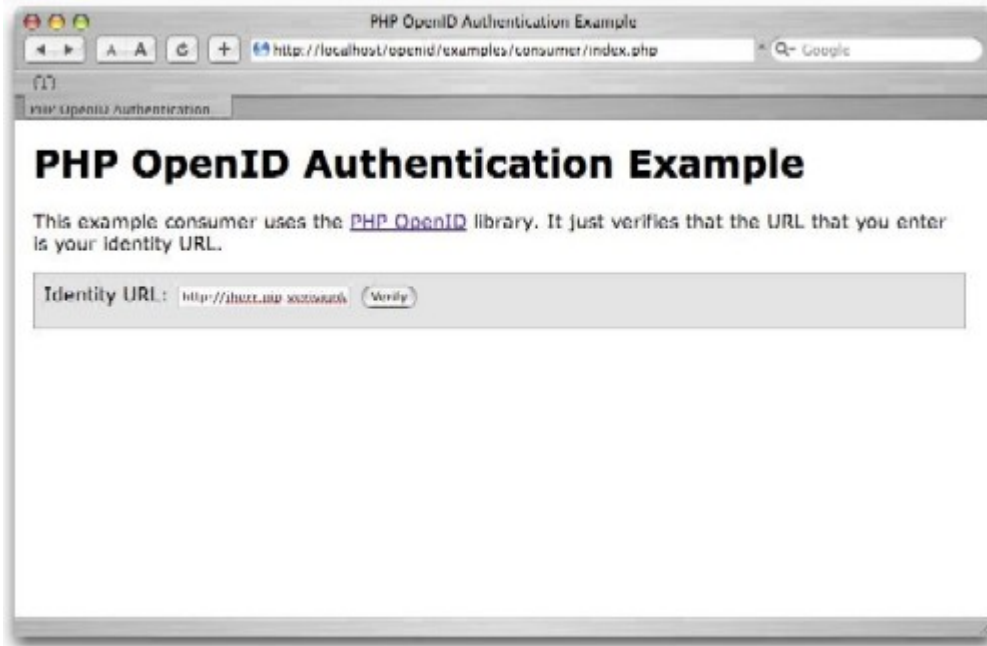
Now, let's take a look at an OpenID consumer library for PHP.

Installing the OpenID Consumer Library

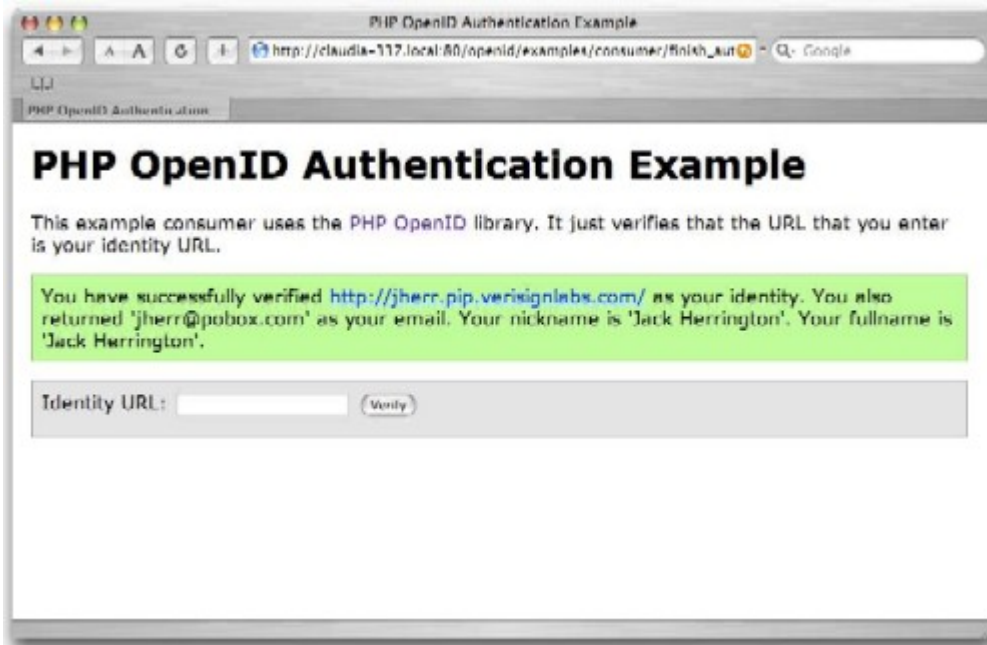
There are a few different OpenID libraries out there for PHP. I picked the first one off the list that I found, which was a library from JanRain. It's located on the OpenID Enabled site. I downloaded the most recent version from the site and then copied that into the documents directory of my Apache server.

Then I ran the Examples/detect.php script at the command line as mentioned at the top of the README file. On my Mac OS X machine with PHP 5 installed it turns out I had almost everything I needed, and what I didn't have was optional. You might find that you have to install GMP, BCMath, or PEAR DB, in addition to enabling the cURL library, and an XML processing extension (like the XML DOM). But all of that is explained in the README file and by the detect.php script.

The README file goes on to talk about installing the OpenID library as a PEAR module, but you don't need to just to try it out. The first thing to do is to go to the consumer test page in examples/consumer.



I type my OpenID as before. I then go to the OpenID provider page and log in. From there I head back to my local example.



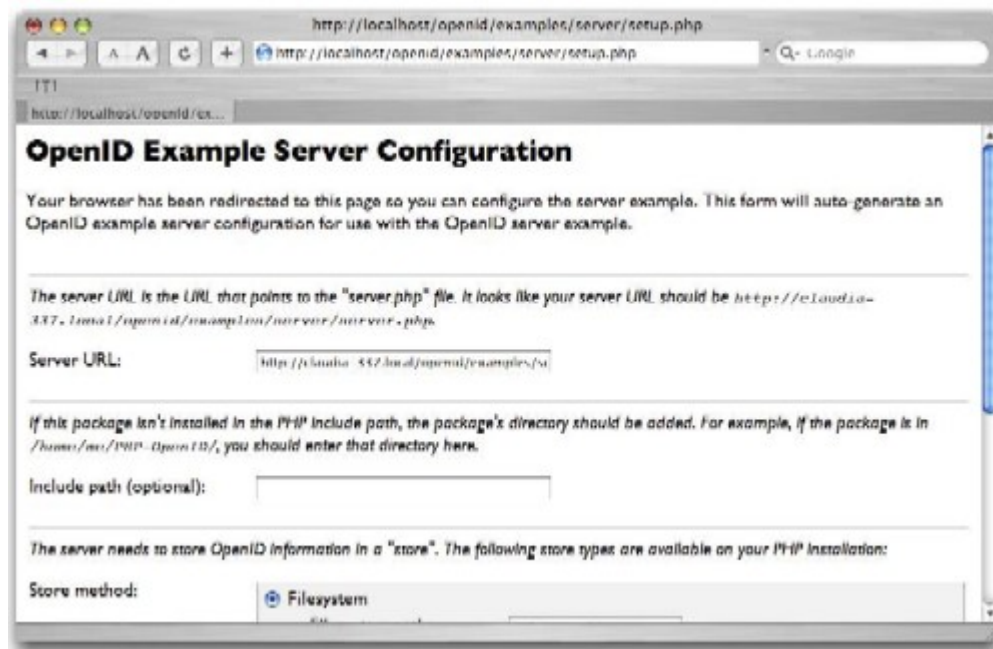
Not only has it received my OpenID URL and validated it, but it has my e-mail address, full name, and nickname as well! Very nice.

You can take a look at the consumer code itself and you will likely use it as a template for implementing your own OpenID consumer application. Frankly, it's a bit more PHP to implement a consumer on this library than I would like. The API is fairly "close to the metal." I'd like to see it be a bit higher level so that there is less code to write to get a consumer site going. But on the upside, all of the error cases are handled and the library is stable and well written.

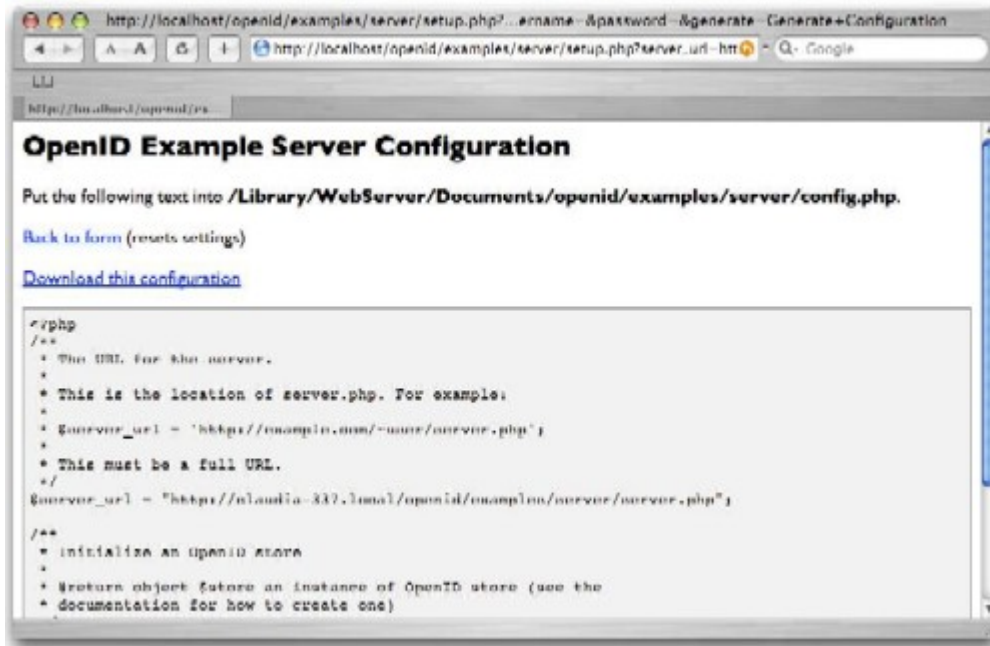
The final step in learning about OpenID at a practical level for PHP is to set up an OpenID provider using an OpenID library for PHP.

An OpenID Provider

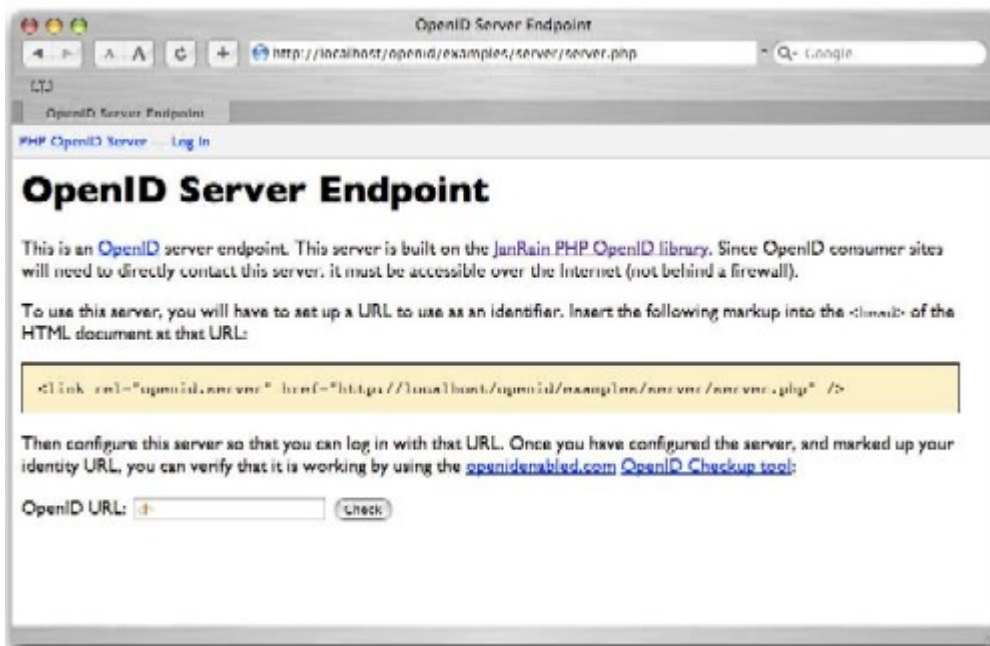
The first step in setting up the provider is to configure it. The provider can use one of several methods to store the associations and so on required to run the provider. It can use the file system or SQL Server. The configuration page is called Setup.php.



When you've set up what works for your site, you get the code for the Config.php file.



After that you take the contents of this page and copy them into Config.php in the examples/server directory. From there you go to Server.php, which gives you some of the basic details of the provider. Most importantly, what to use as the `<link>` in your own pages to have this provider support your OpenIDs.



From here I can build my own OpenID with this provider by adding `/userpage/user=<username>` to the end of the provider URL. This returns an identity page.



That's the same kind of identity page that other services, like the Verisign one that I used at the start of this article, return for OpenID services. This page is shown as HTML in Listing 4.

Listing 4. The user page returned from the provider

```
<html>
<head>
  <link rel="openid2.provider openid.server"
href="http://localhost/openid/examples/server/server.php/userXrds?user=jherr" />
  <meta http-equiv="X-XRDS-Location"
content="http://localhost/openid/examples/server/server.php" />
</head>
<body>
  This is the identity page for users of this server.
</body>
</html>
```

Because I've installed this on localhost the OpenID provider that I installed isn't going to do much good on the Web. From here I need to install this on my Web site, get it hooked into my database, and use it from there.

The consumer and provider examples provided here serve only as templates on which you can base your own work. You could, for example, use the OpenID consumer example to add the ability to log in using an OpenID URL as an account to your online services. Or you could use the provider example to provide OpenIDs to all of the users on your service for free.

Where to Go Next

So far in this article I've talked about what OpenID is and what it can do, but I really haven't recommended it for any particular types of sites. While there are no specific rules, I would say that OpenID is not strong enough of an identity and security mechanism to trust to financial data. So I wouldn't use it in e-commerce applications. But there is a lot more on the Web than financial sites.

The OpenID directory provides an amazing list of sites that are open to those with OpenID identities. These include blogs, discussion boards, social networks, wikis, even sites that do local event meetup type things. The Web is inherently multi-user, which means we have to keep track of identities. But not all sites are created equal in terms of the strictness of identity requirements. Just compare FaceBook to Wells Fargo to get an example of what I mean.

To learn more about OpenID I strongly recommend going to OpenID.net. That site has another good introduction to OpenID, as well as the OpenID specifications and a list of consumer and provider code written in all of the popular programming languages. An OpenID book is also available free online. The book goes into great detail on how the protocol works and shows each of the HTTP requests as they appear over the wire so you can follow the whole authentication process step-by-step.

OpenID is an identity framework with a lot of potential that many companies, Web sites, and organizations have already bought into. AOL, as an example, has built OpenID support into its Open Authentication API (OpenAuth) services, which extends OpenID to any site that uses OpenAuth for authentication. In fact, one of the easiest ways to support OpenID is to use OpenAuth from AOL and you also pick up AIM and AOL users in one fell swoop. All in all, I'd say OpenID has a bright future ahead of it. It makes it easier to use more of the services on the Web, and that's a win for everyone.